



National University of Ireland, Maynooth
Department of Electronic Engineering

NUI MAYNOOTH EE297 – Intelligent Systems Project
Óllscoil na hÉireann Má Nuad

2nd Year Project: Can Detecting Neural Network

Team 1: Insert Team Name Later

Students:

Cael Timmons, 19451342

Heather Bruen, 19356061

James Florin Petri, 19712119

Table of Contents

1. Abstract	2
2. Declaration	2
3. Introducing the problem	3
4. Theory	4
4.1. Basics.....	4
4.2. Neural Networks in more detail.....	5
4.3. Types of Neural Networks.....	10
4.4. Training a Neural Network.....	12
4.5. TensorFlow Neural Networks.....	14
5. Implementation	15
5.1. Overview.....	15
5.2. Gathering images.....	15
5.3. Labelling images.....	18
5.4. Preparing the environment.....	19
5.5. Training and optimisation.....	20
6. Results	24
6.1. General Results.....	24
6.2. Quantifying the results.....	25
6.3. Final Results.....	27
7. Conclusion	28
8. References	29
9. Appendix	30
9.1. Tables	30
9.2. Python Script – Pictures	33
9.3. Python Script – Webcam.....	35

1. Abstract

The objective of this report is to document and present the techniques used for analysing, designing, and improving a neural network, with the goal of creating a successful can detector. The approach followed in the project revolves around creating a convolutional classifying neural network by using the TensorFlow API in concordance with CUDA 10 in an Anaconda environment with a custom-made dataset. In this report, the different choices and actions the authors made throughout the process of making this neural network will be covered and explained. This report will also explore the theory aspect of Neural Networks with the intent of opening this report up for readers that are not necessarily involved in the subject. The process on implementing the network and showcasing its code will be discussed along with the various tools that were needed to make a neural network that can recognize cans.

2. Declaration

By signing this document, I declare the following:

- That I have read and understood the Departmental policy on plagiarism.
- That this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institution of tertiary education.
- That information derived from the published or unpublished work of others has been acknowledged in the text and a list of references is given.

Signature team member 1:



Signature team member 2:



Signature team member 3:



Date: 9th of May 2021

3. Introducing the problem

The task set for this project was to document the creation of a neural network. An application of a neural network was chosen, the types of networks most suited for that purpose were researched and the network was implemented. Neural networks have many applications, but long story short, they take an input and pass it through layers that multiply it by different weights and biases to give an output. These weights and biases are set during training where the network is given labelled examples of inputs, based on which the said weights and biases are changed until the outputs match the labels.

The application chosen was to create a neural network that can recognize drink cans. The idea was that for any image or video fed into the network, it should identify and outline any drink cans present. This application was chosen as it could help solve the problem of littering and pollution by helping with the aspect of recycling aluminium cans. The idea being that two possible practical application of this neural network could be utilized, the sorting of waste in a recycling centre and the collecting of waste by an automated cleaning robot.

Before anything else, this report will talk about the theory behind neural networks. Then it will go into detail about how they work, what the different types are and what tasks each type is designed to perform. Then it will explain which type was chosen and the reasons why that type was chosen over the others. Afterwards the implementation of the network is described, showing how it was coded and how the various tools used were setup. Next, the creation of a custom dataset that used to train the network is chronicled. The types of images used and why they were used is explained. Subsequently it shows how the accuracy and reliability of the network was tested and the results of these tests. Finally, it gives conclusions on this process, what the network does well, what could be improved, how well the project was executed and takeaways for future projects.



(Fig.1 – Waste exemplification)

4. Theory

Neural Networks can be quite complicated and hard to understand, so in order to make the report more accessible to everyone, in this section the theory aspects of Neural Networks will be covered.

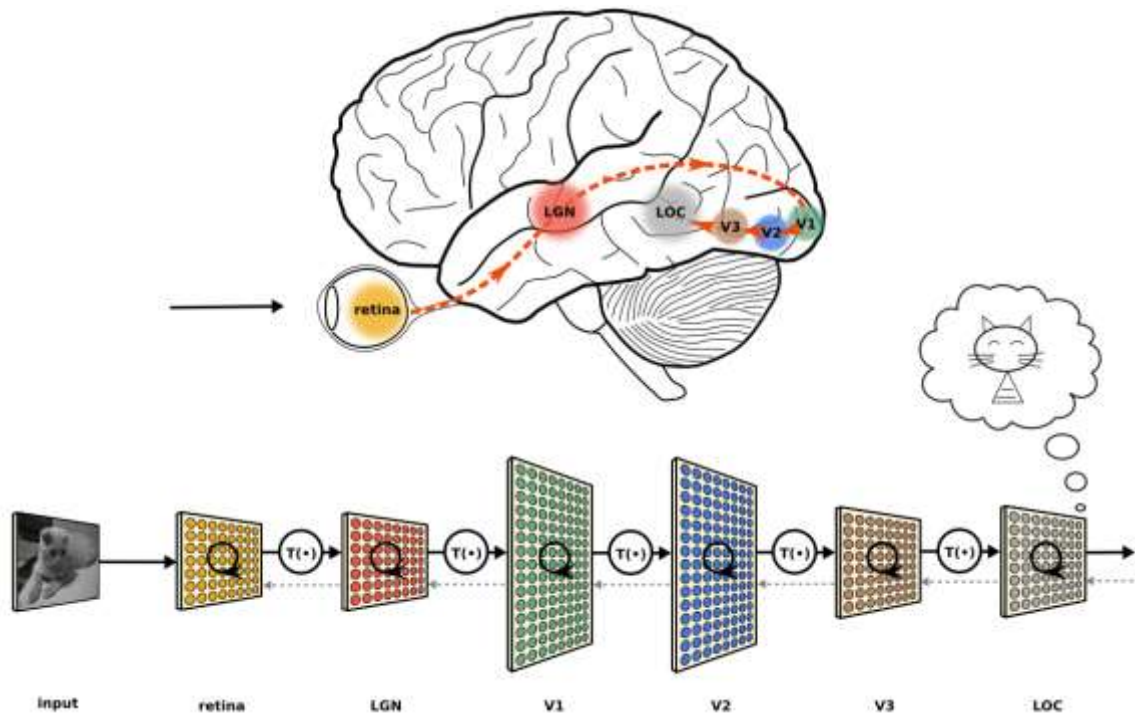
4.1. Basics

In order to understand the idea of a Neural Network, it is enough to look at biology. If the brain was to be explained, a simple way to do so is by calling it a network of neurons that are changing their output by the electrical intensity and chemical structures pumped into them. The same principle holds for Artificial Neural Networks (ANNs) which can be considered to be a software equivalent of a neural structure where the neurons act more like switches that provide an output for the inputs it receives. Just how an organic neuron's output can be the input to thousands of other ones, which can be said about the artificial neurons as well.

Teaching a neural network might seem cumbersome at the start as when certain new information is learned, certain neural connections are activated more frequently, leading to stronger connections, and therefore increasing the chance for the artificial neural network to produce the desired output. This is similar to how humans learn, keeping the biological brain analogy. Take a baby for example. The said baby starts by knowing almost nothing but by repeatedly trying different experiences the baby learns from the information it is given. If a baby is never shown an aluminium can, it will have problems understanding what it is at the start, but by playing with it and experiencing it, he/she learns that new information. This repeating learning cycle is done through a feedback structure until the desired outcome strengthens the specific neural connections.

Humans are well known for their ability to learn by observing, but also learn by experimenting. This also translates to Artificial Neural Networks as the training can be narrowed down to two main methods: supervised and unsupervised. In unsupervised training the ANN is given the input information and it has to attempt to learn and "understand" it "on its own". On the other hand, in a supervised ANN, the network is trained with matched/similar input and output data samples, with the intent that the training will ultimately provide a desired output for all inputs of a certain trained situation. For example, in a can detector, images of cans will be provided to train and then test its accuracy and improve by experimenting over and over again [1].

As it could be seen from this section neural networks are a simplified model of a brain's behaviour which can be implemented and adapted in different ways. For better visualization, a diagram can be seen in Figure 2 below of the two compared.



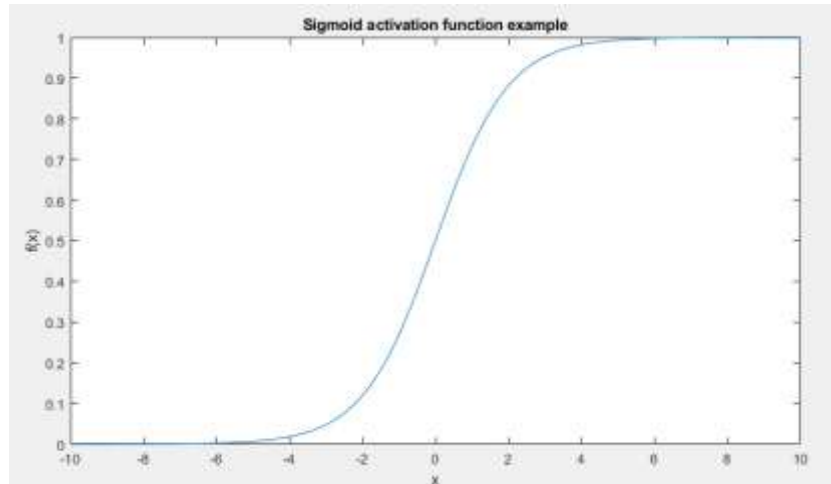
(Fig.2 – Visual analogy of a neural network as a biological system)

4.2. Neural Networks in more detail

Based on the basic image the previous step created, the theory and details of neural networks will be covered in this section. What was previously called a neuron is described in the ANN by an “activation function”. An activation function works similar to a switch, where it can change between states such as 0 to 1, from -1 to 1 and so on. The most commonly used activation function is the sigmoid function, equation of which can be seen in Formula 1 and its plot can be seen in Figure 3, both below this paragraph. Different activation functions will be covered later in the section as the sigmoid function will be used to explain the main concepts of a neural network. It should be noted that, even though MATLAB was not used to implement the Neural Network created in this project, it will be used to easily explain the theoretical concepts behind them.

$$f(x) = \frac{1}{1 + e^{-x}}$$

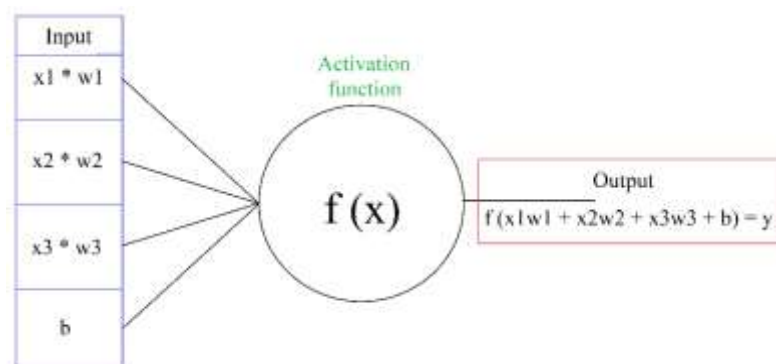
(Formula 1 – Sigmoid activation function)



(Fig.3 – MATLAB simulation of a basic sigmoid activation function)

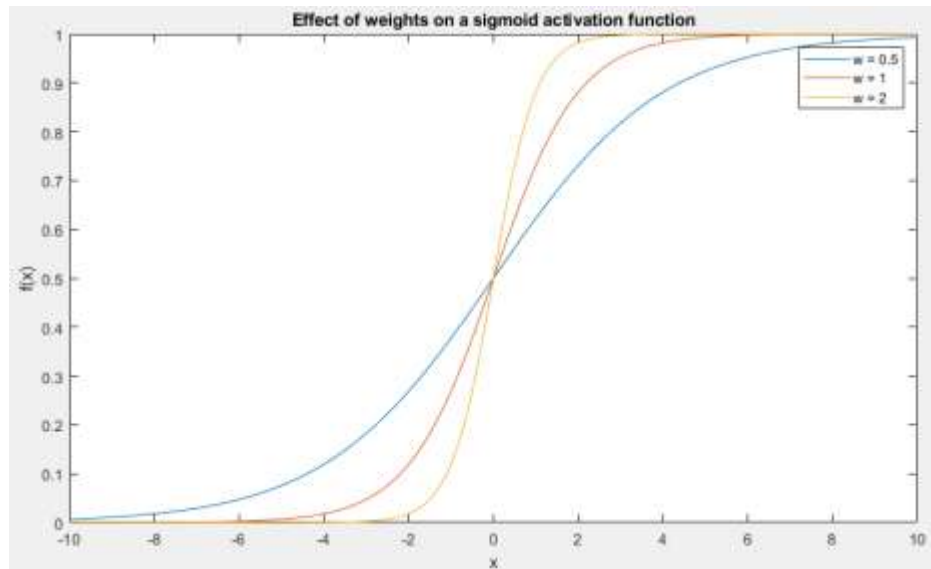
As it can be seen from Figure 3, the function is activated when the input is greater than a certain value (activated i.e., moves from 0 to 1). As the sigmoid function is not a step function, the transition band is gradient.

Just how the biological neurons are interconnected, an artificial neural network can be represented by organized layers of nodes, nodes which take multiple weighted inputs and apply the activation function to the summation of the inputs in order to generate the appropriate output. In Figure 4 below, a node can be seen with 4 inputs, 3 variable inputs with weights applied to each of them and one bias input (Will be covered later in the section).



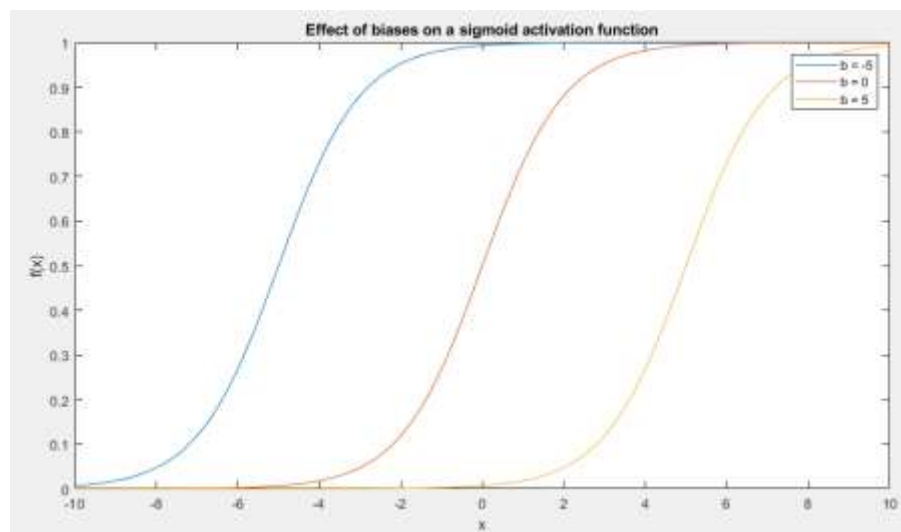
(Fig.4 – Structure of a node)

Weights are variables that are changed during the training process which along with the input form the output. In Figure 5 below, the sigmoid activation function was adapted for a single weighted input in order to present the effects of the weights on a node.



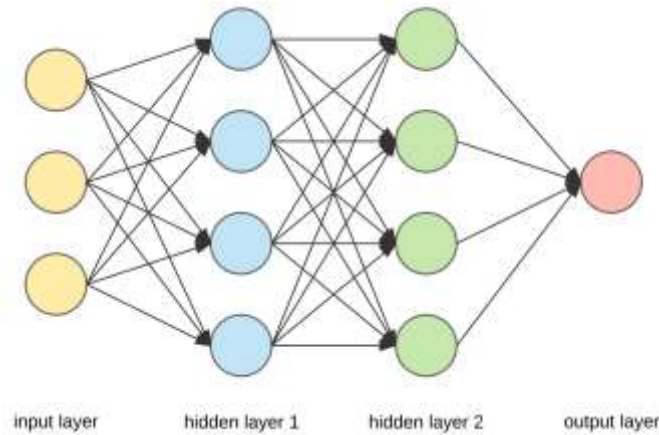
(Fig.5 – Effects of weights on a single input activation function)

It can be seen that the weights affect the slope of the output of the activation function, which is used in order to represent different relationships between inputs and outputs by strength. Meanwhile biases add flexibility to the node by affecting the transition areas center location as it can be seen in Figure 6 below. It should also be noted that the weights and the biases hold real values (non-binary).



(Fig.6 – The effects of biases on a single input activation function)

Because the bias decenters the function from 0, as it can be seen in Figure 6 above, it is common to use biases as if statements because the affect when the node activates in the form: if ($x > z$) then 1 else 0, where z is affected by the bias.



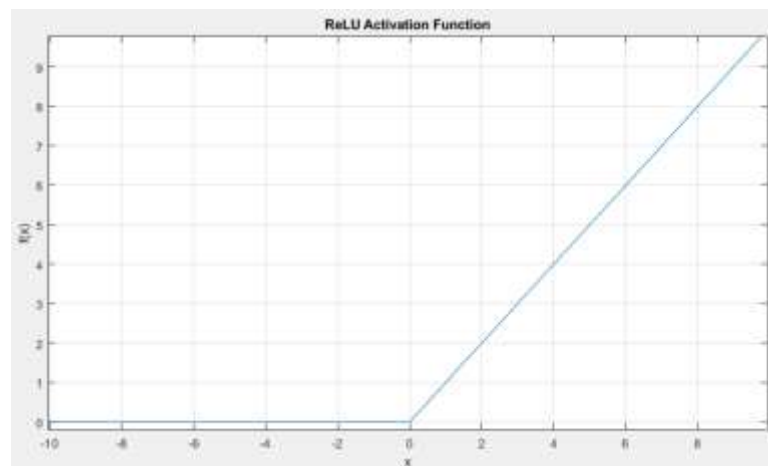
(Fig.7 – A Neural Network's basic structure)

By looking at a neural network's basic structure in Figure 7 above, everything starts to come together. The last thing to cover regarding the basics is the concept of layers. Layers are groupings of nodes operating at a specific depth in the NN. For a basic neural network, there are 3 types of layers: the input layer (the initial data that needs to be processed by the NN), the output layer (the final result that the given inputs produce) and the hidden layers (layers between the input and output layers where all the computation is done).

Up until now, the sigmoid activation function was used to explain everything, but in the project a Rectified Linear Units (ReLU) activation function was used, which was optimized by the Faster RCNN with Inception-V2 Architecture. ReLU are commonly used in Deep Neural Networks (DNNs) as activation functions, but they can also be used for Convolutional Neural Networks (CNNs). The specific formula and plot can be seen below in Formula 2, respectively Figure 8.

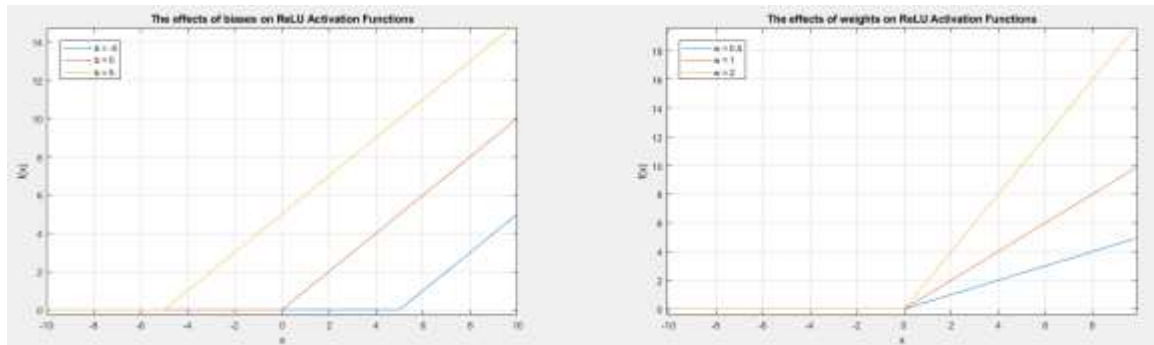
$$f(x)=\begin{cases} 0, & x < 0 \\ x, & \text{otherwise} \end{cases}$$

(Formula 2 – Basic ReLU activation function)



(Fig.8 – Visualization of a ReLU activation function)

As it can be seen from Figure 8, the function stays at 0 until the trigger point ($x = 0$) when it starts increasing linearly ($f(x) = x$). The effects of weights and biases have a similar effect on the function to the one of the sigmoid activation functions as it can be seen from Figure 9 below.



(Fig.9 – Effects of weights and biases on a ReLU activation function)

There are many more types of activation functions, but the main ones are covered in the table below in Figure 10.

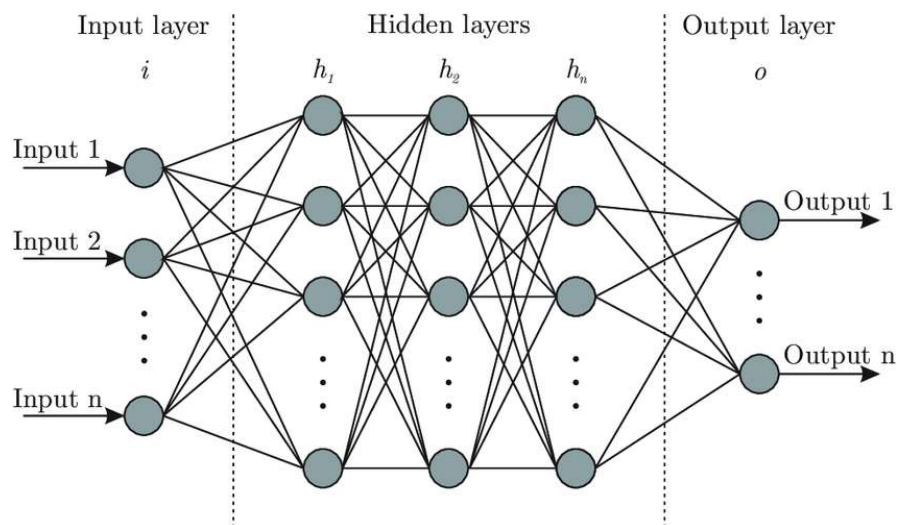
Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
Tanh		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parametric Rectified Linear Unit (PReLU) [21]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [21]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

(Fig.10 – Most common activation functions)

4.3. Types of Neural Networks

Before delving into the nature of the neural network for this report, the different types will be further explained along with each of their prospective advantages and disadvantages. Recurrent Neural Networks (RNN), Artificial Neural Networks (ANN) and Convolutional Neural Networks (CNN) are the three types that will be focused upon here. Neural networks have four main characteristics, including being non-linear, non-limited, non-qualitative and is non-convex [2]. Non-linearity's importance in a neural network requires no further explanation as this is what allows mapping between the neurons occur. A neural network should not be limited, as it needs to allow as many neurons to function. Neural networks should also be non-qualitative. This means that it has the ability to learn and change. Finally, non-convexity means that a network would have a vast selection of extrema, therefore, it leads to a diverse system evolution.

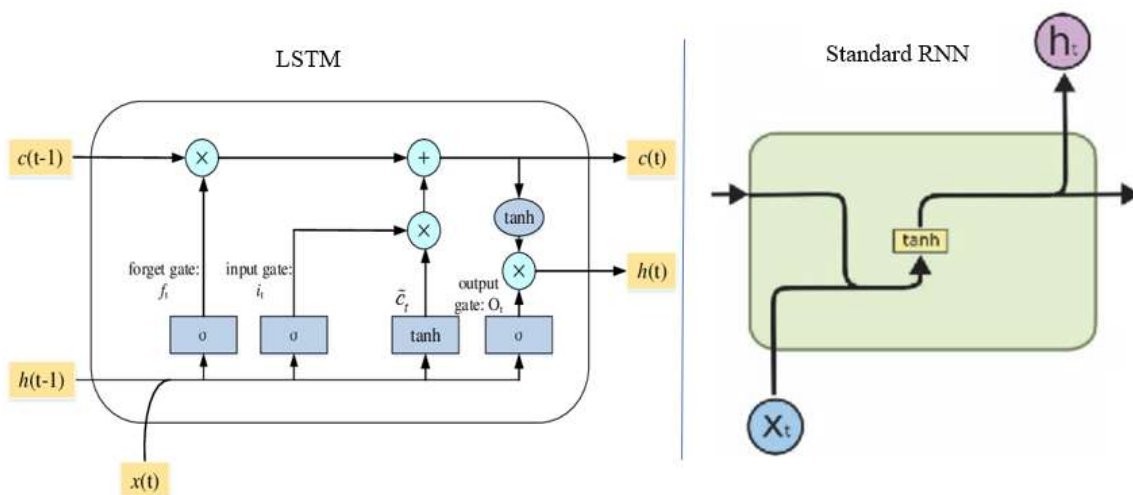
Artificial Neural Networks take their inspiration from the human brain in the way it has neurons and connections between each node representing a weight. The neurons in neural networks are very similar to biological neurons where information is transmitted by electrical signals from dendrites to axons. An Artificial Neural Network is a group of multi-layered perceptrons and works as a feed forward neural network. ANNs consist of three main layers: the input layer, the hidden layer, and the output layer as seen in Figure 11. The input layer primary role is to take in data and pass it to the hidden layer. The output layer predicts and classifies the data and finally the hidden layer between both layers is where most of the work is done as it extracts what it deems useful features which allow the output to predict values [3]. Due to the feed forward method of ANNs, choosing this to be basis of image classification is not ideal. ANNs also work primarily with one-dimensional vectors making two-dimensional images even harder to work with.



(Fig.11 – Layers of an Artificial Neural Network)

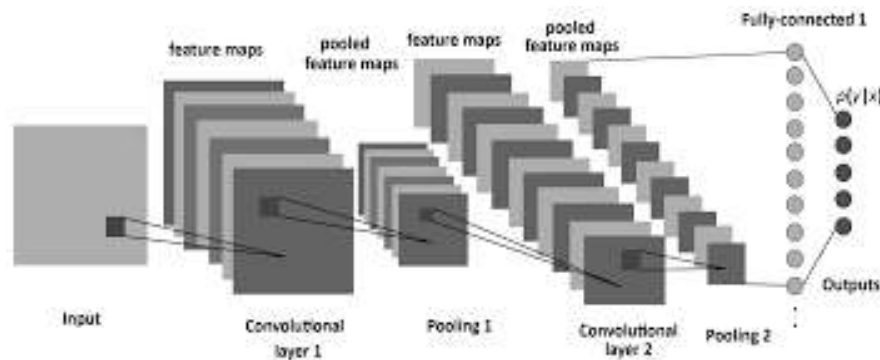
One issue with this is that the spatial features are lost in an image. Another problem associated with artificial neural networks is the vanishing and exploding gradient problem. This means that a gradient will grow until it gets to the point of exploding due to derivatives being multiplied together. Vanishing gradients is the opposite as the derivatives get smaller and decrease to a point of vanishing. This problem can make the training process much slower as the loss function would be quite poor [4]. Due to this, using convolutional neural networks or recurrent neural networks are much more efficient.

Recurrent Neural Networks operate on an input space and on an internal state space by using what has already been processed by the network. These neural networks have the name recurrent as they are looped and make the same operations for each element in its given data. RNNs are proficient at modelling sequential data such as time series or language [5]. Recurrent neural networks not only take the input into consideration but also the previous inputs which it allows itself to memorize what has previously happened. Recurrent Neural Networks are generally used for prediction, by this it is meant that text, audio, stocks are where RNNs are likely to be used. This type of neural network works with Long Short-Term Memory architecture or LSTM as it is also known. The original idea behind LSTM was to prevent the vanishing and exploding gradient problem that was mentioned above. A LSTM unit is composed of an input gate, an output gate and a forget gate. The forget gate allows a network to reset its state [6]. These gates determine what information is deleted and what information gets to influence the output. The standard RNN has a single layer whereas LSTM has four interacting layers as seen below in Figure 12.



(Fig.12 – Long Short-Term Memory vs Standard RNN)

Finally, the chosen type of neural network for this project was Convolutional Neural Network. CNNs can be applied to two dimensional arrays such making it the ideal type for dealing with images. This network gets its name from its many hidden layers. Within a CNN there are pooling layers, fully connected layers, and convolutional layers within the hidden layer. The pooling layers aim is to reduce the dimensions of a feature map. Pooling is a process where samples are made by pool operators scanning and collecting information [7]. The main goal of the convolutional layer is to learn feature representations as inputs. This layer applies weights to the input sections from an image in order to create feature maps. The final layer is the fully connected layer. This layer aggregates all the neurons from the last layers and connects them to each neuron in the current layer [8]. These 3 layers can be seen in Figure 13.



(Fig.13 – Layers of a CNN)

Convolutional neural networks are more suited to spatial data over the previous two types mentioned above. This is due to ability it has to filter across special dimensionality to make a two-dimensional activation map. After an image goes through the process of the above layers, the CNN should be able to determine key features as an image is at this point reduced to only this. In order to train convolutional neural networks, backpropagation is the most common method as it works with a gradient of a loss function to determine how to adjust in order to improve performance. This made CNN the ideal choice for this project for the classification of cans.

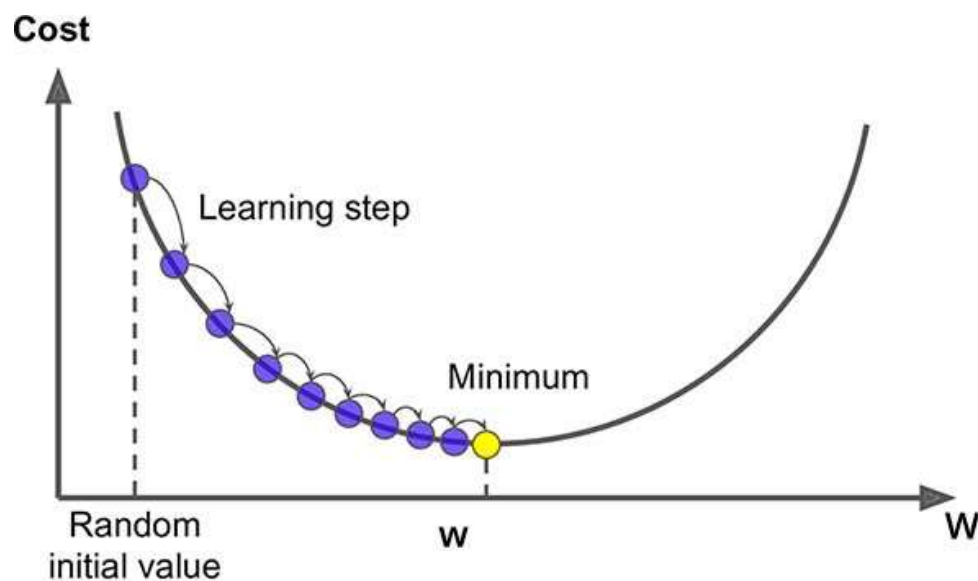
4.4. Training a Neural Network

The goal of training a neural network is simple, to get the network to understand the relationship between the inputs supplied to it and the outputs desired from it. This is so that when presented with an input it should be able to give a correct output on its own. This is done by changing the weights and biases of the hidden layers within the network until the network can accurately give the correct output to a given input. So how is this done?

The first step in training a neural network is the creation of a dataset. The dataset contains numerous examples of inputs to the network, each paired with a label of what output that input should result in. The dataset should contain variations of each input so that the network can recognise the input in multiple presentations. The process of pairing each input example to the corresponding output label is usually done manually. Considering that datasets usually consist of hundreds of examples this is a tedious process, as such, many try to use existing datasets to avoid having to perform this process. However, if the application of the network is quite specific, such datasets may not exist and so a custom dataset must be made, just as in this project's case.

To start the weights and biases of the network are initialized with random values. Then the dataset is run through the network where the difference between the given outputs and the desired outputs is determined. This is done using a loss function. The loss function subtracts the given output from the desired output thereby giving an error value. The larger this error value then the more the network needs to change the weights and biases. The process is repeated until the loss function value is below a certain threshold, the threshold being determined by how accurate the network needs to be. The network is then considered trained.

The loss function is minimized by a gradient decent algorithm. This algorithm finds the global minimum of the loss function similar to how a ball in a valley will roll to the bottom, which is exemplified in Figure 14. The algorithm picks a random spot along the function and finds the derivative. The derivative can be thought of as the slope of the function and so the algorithm moves down the slope until the minimum is found. This 'negative gradient' is then used to adjust the weights and biases of the network so as to reduce the negative gradient. This process is called back propagation. [9]



(Fig.14 – Gradient decent exemplified)

4.5. TensorFlow Neural Networks

TensorFlow is a well-known open-source deep learning framework that is developed and maintained by Google, logo of which being seen at the end of the section. Technically, “TensorFlow uses dataflow graphs to represent computation, shared state, and the operations that mutate that state” [5]

TensorFlow was chosen for this year’s project as it deals with much of the complexity behind forming a neural network, letting the user focus on experimenting, and perfecting the model. TensorFlow provides the modeller on its dedicated GitHub page, organized by version and offering a variety of open-source configuration and operation alternatives right out of the package.

Some other advantages that TensorFlow brings are very good visualization tools (TensorBoard), easy debugging (good error messages, error pointers, running subparts of graphs for problem detection, etc.), scalability (based on the configuration chosen, the final model can be implemented to everything from a high-end computer to a smartphone) and pipelining (parallel design and GPU support).

Like everything, TensorFlow has disadvantages as well, some of them being: Linux focused environment (Windows support is bad at times, but there are ways to go around it, such as using the anaconda environment), slow training and resource intensive, only NVidia GPUs supported for GPU training.

Overall TensorFlow suits the implementation for this year’s project as a high-end machine with an NVidia GPU was available, leaving only the installation of software and preparing and debugging the environment for use.



(Fig.15 – TensorFlow logo)

5. Implementation

The process in which Neural Networks are made can be quite simple to follow but also difficult if not done correctly. In this section, the process this project follows is described in sections below.

5.1. Overview

The implementation of this convolutional neural network for it to be able to detect cans is where the main results can be taken. In order to make a neural network successful, the implementation must be done well. The process in which this neural network was built will be discussed further under each of their respective sections.

Regarding the image database, the way in which it was built will be split into the gathering of all the images and how the pictures were labelled to be used for this classifying neural network. After the image database is made, a suitable environment was necessary to be prepared. Once the environment was ready and the images were all labelled, the training process was ready to begin. A label map is made to help the trainer to identify objects by defining a mapping of class names. The training is the final step to ensure the neural network is optimized.

5.2. Gathering images

The first step in creating the dataset was the gathering of images to use in the dataset. These pictures had to be varied enough so that the network could detect any can presented to it. As a result, numerous images were required. This dataset contained about 800 images by the end of the training process.

Initially the images were just of different types of cans; beer cans, soft drink cans, 330ml cans and 500ml cans. These images were taken from as many different angles as possible to ensure that the network could recognise any type of can from any angle. These images were also taken against numerous different backgrounds such as: concrete, tarmac, grass, dirt, gravel, wood and carpet. This, again, was in an effort to make the network as versatile as possible.



(Fig.16 – Examples of Initial Dataset Images)

Next, images were taken of deformed cans. Again, numerous types of cans were used and the pictures were taken on many different backgrounds. The cans were initially only slightly deformed and pictures taken from many angles. Then cans were then gradually deformed more and more, pictures taken at each step. This was to ensure that the network could recognise a can in any state of deformity.



(Fig.17 – Examples of Deformed Can Images)

Subsequently images were taken of cans in various states of obturation. Again, numerous types of cans were used and in various states of deformity. Various states of obturation were used such as cans in long grass, cans buried in gravel, cans held in a hand, cans in front of other cans and cans partially hidden beneath other objects. This was again an attempt at making the network more versatile.



(Fig.18 – Examples of Obturated Can Images)

Finally, images were taken with cans next to other objects. This was done in an attempt to prevent false calls. As before, numerous types of cans in different states of deformity on different backgrounds were used. Many objects were used to accompany the cans such as: glass bottles, plastic bottles, trays, cardboard and humans.



(Fig.19 – Examples of cans next to other objects)

After the images were gathered, they were split in a 1:4 ratio into test and train images. It was ensured that images from the steps mentioned above were distributed between the test and train images. The train images were used to, unsurprisingly, train the network while the test images were used to evaluate the performance of the network. Validation images are used to tune the networks hyperparameters such as the number of hidden units. However, they were not used, as the parameters were already set in the example that this network was based off. [10]



(Fig.20 – Test vs Validation)

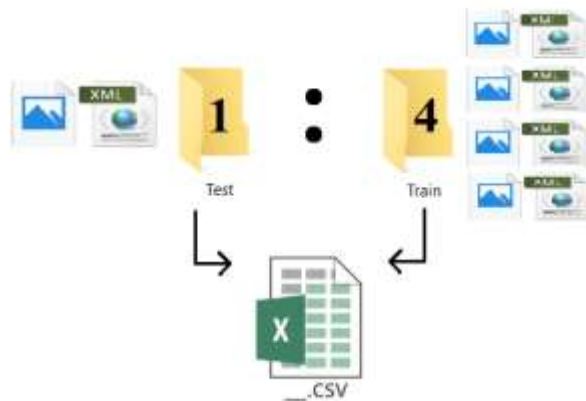
5.3. Labelling images

Once a vast enough selection of pictures was gathered, the process of labelling could begin. Around 800 images with varying amounts of cans had to be labelled, allowing this neural network to become more robust. This part of the implementation was done with LabelImg, a graphical image annotation tool. After adding a directory to the folder holding the images of cans, the labelling process was simple although time consuming. Within each image, the user drew a box around the object they wished to label. The user then added the label they wish to use and saved this as an .xml file. This can be seen in Figure 21. This .xml file was added to the same folder holding the images, which was then used to generate one of the inputs to the TensorFlow trainer.



(Fig.21 – Labelling a can in LabelImg)

After all the images were labelled, the images and their respective .xml files were split into a ratio of 1:4 for testing and training images. The training images were used to make the neural network more robust and the images in the test folder to be used as a way to assess this project. The label parameters were then saved in .csv form in order to condense the information for the training process. The diagram in Figure 22 shows this process on the right.



(Fig.22 – Diagram of labelling process)

5.4.Preparing the environment

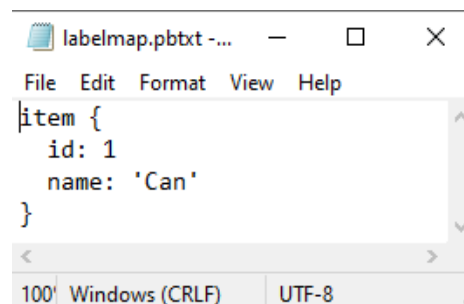
After the labelling process, this was the most dreadful part of the project. The main problem came from incompatibility issues between TensorFlow, CUDA and cuDNN.

The first step was preparing the environment. This started by installing Python 3.9 and Anaconda 4.9.2. Although Anaconda isn't a requirement for TensorFlow object detection API, because it is mostly Linux centric, the Anaconda environment offers the tools needed to ease the use in windows and it also offers easy to use virtual environments.

The next step was figuring out the compatibility versions between TensorFlow, cuDNN and CUDA as the newer versions are not backwards compatible and some packages required for older versions are version locked or in some cases even deleted. In the end after experimenting with a few versions, TensorFlow 1.13 with CUDA 10 and cuDNN 7.4 were chosen. It should be noted that the version for python that was recommended was 3.7, but the machine used for training the model had another project that was locked on 3.9. For this reason, with some manual modifications to the automatically installed packages, the model was adapted for python 3.9 with no negative effects except for some warnings that can be ignored.

With the environment set up and all the pictures are labelled, organized and condensed, the next step was preparing the trainer. For this, the label map and configuration files needed to be implemented, with some intermediate steps that are done with simple console commands like forming the proto equivalent files.

Because the neural network has to identify only the cans, the only label required is the can label. One thing to note is that the label map has to be of the format .pbtxt and not .txt as it will not work otherwise. For this reason, the label map is as simple as it can be seen in Figure 23.



(Fig.23 – Label map)

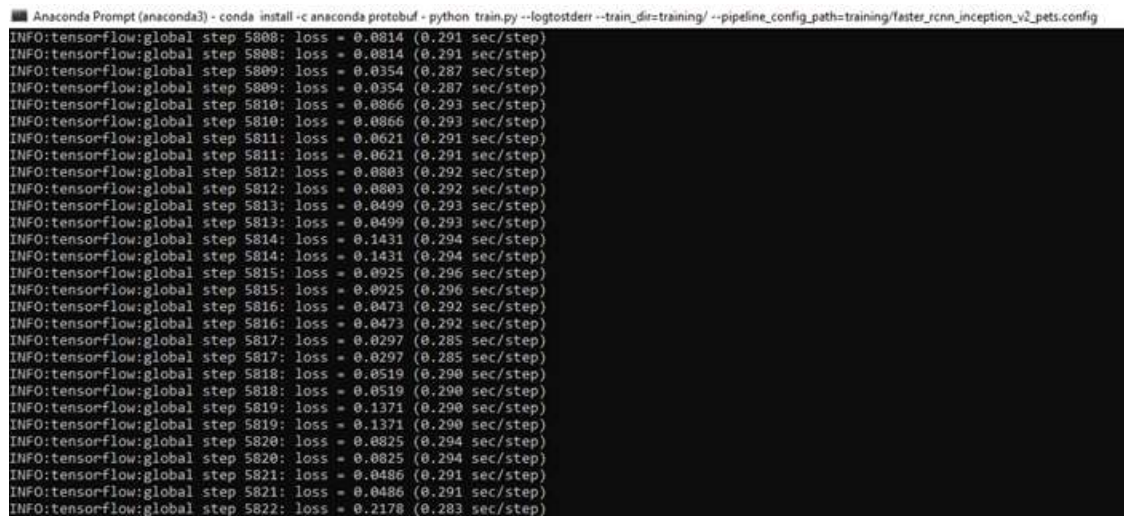
After a bit of experimenting with creating a custom-made configuration file, the conclusion that it is very time consuming to make it efficient came very fast. For this reason, the Faster R-CNN with Inception V2 configured for the Oxford-IIIT pets' dataset

was taken and modified (It is open source). The can detector's parameters and file organization were inserted to the config file, along with the training data information and details.

With everything configured and ready for the training, the only step remaining was the actual training in order to see the results.

5.5. Training and optimisation

With everything set up and working correctly, the trainer was initialized. After around 30-60 seconds, TensorFlow will start the training process while displaying in the console the step count, the time required for the step to be computed in seconds and the loss. A screenshot of the training process at around the 5000th step can be seen in Figure 24 below.



```
Anaconda Prompt (anaconda3) - conda install -c anaconda protobuf - python train.py --logtostderr --train_dir=training/ --pipeline_config_path=training/faster_rcnn_inception_v2_pets.config
INFO:tensorflow:global step 5808: loss = 0.0814 (0.291 sec/step)
INFO:tensorflow:global step 5808: loss = 0.0814 (0.291 sec/step)
INFO:tensorflow:global step 5809: loss = 0.0354 (0.287 sec/step)
INFO:tensorflow:global step 5809: loss = 0.0354 (0.287 sec/step)
INFO:tensorflow:global step 5810: loss = 0.0866 (0.293 sec/step)
INFO:tensorflow:global step 5810: loss = 0.0866 (0.293 sec/step)
INFO:tensorflow:global step 5811: loss = 0.0621 (0.291 sec/step)
INFO:tensorflow:global step 5811: loss = 0.0621 (0.291 sec/step)
INFO:tensorflow:global step 5812: loss = 0.0803 (0.292 sec/step)
INFO:tensorflow:global step 5812: loss = 0.0803 (0.292 sec/step)
INFO:tensorflow:global step 5813: loss = 0.0499 (0.293 sec/step)
INFO:tensorflow:global step 5813: loss = 0.0499 (0.293 sec/step)
INFO:tensorflow:global step 5814: loss = 0.1431 (0.294 sec/step)
INFO:tensorflow:global step 5814: loss = 0.1431 (0.294 sec/step)
INFO:tensorflow:global step 5815: loss = 0.0925 (0.296 sec/step)
INFO:tensorflow:global step 5815: loss = 0.0925 (0.296 sec/step)
INFO:tensorflow:global step 5816: loss = 0.0473 (0.292 sec/step)
INFO:tensorflow:global step 5816: loss = 0.0473 (0.292 sec/step)
INFO:tensorflow:global step 5817: loss = 0.0297 (0.285 sec/step)
INFO:tensorflow:global step 5817: loss = 0.0297 (0.285 sec/step)
INFO:tensorflow:global step 5818: loss = 0.0519 (0.290 sec/step)
INFO:tensorflow:global step 5818: loss = 0.0519 (0.290 sec/step)
INFO:tensorflow:global step 5819: loss = 0.1371 (0.290 sec/step)
INFO:tensorflow:global step 5819: loss = 0.1371 (0.290 sec/step)
INFO:tensorflow:global step 5820: loss = 0.0825 (0.294 sec/step)
INFO:tensorflow:global step 5820: loss = 0.0825 (0.294 sec/step)
INFO:tensorflow:global step 5821: loss = 0.0486 (0.291 sec/step)
INFO:tensorflow:global step 5821: loss = 0.0486 (0.291 sec/step)
INFO:tensorflow:global step 5822: loss = 0.2178 (0.283 sec/step)
```

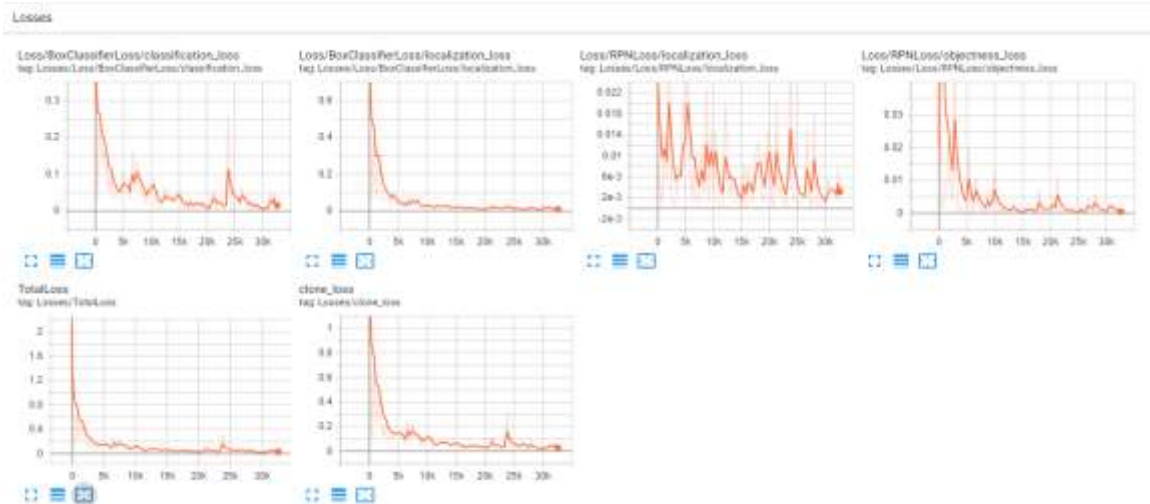
(Fig.24 – TensorFlow trainer console display)

Basically, each step of the training computes and reports the loss. It starts high and it gets lower and lower as the training goes on. The loss is the value calculated by the loss function which is used to “maximize or minimize the objective function, [representing the search] of a candidate solution that has the highest or lower score respectively” [11]. In this project's case the minimizing of the objective function is the focus as the error has to be as low as possible. The training process was manually stopped when the loss was consistently under 0.05 which is the recommended value for the Oxford Faster R-CNN with Inception-V2 variant model, which happened after around 3 hours of training and 31806 steps.

An important point to mark is that the team was lucky enough to have an NVidia GPU, as the TensorFlow GPU training feature is available only on NVidia cards. The card that

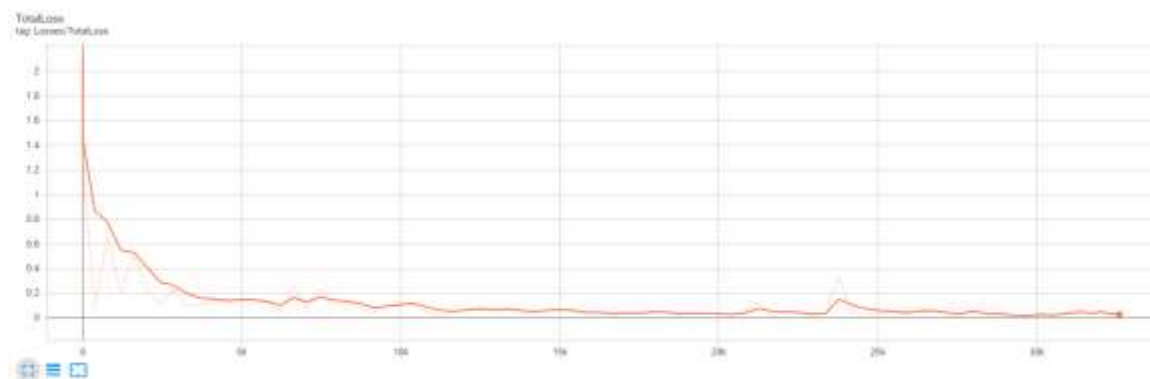
was used was an NVidia GTX 1650 4 GB laptop card, which has a computation capability score of 7.5, the high computing capability score making the 3 hours training time possible.

The training process could be followed using the TensorBoard as well, which offered a more visual representation about the evolution of the model over time. The team's board has many loss function follower plots that are included with the stock TensorFlow API, but the only ones that are used are the objective loss, the classification loss, and the total loss. The TensorBoard loss followers can be seen below in Figure 25.



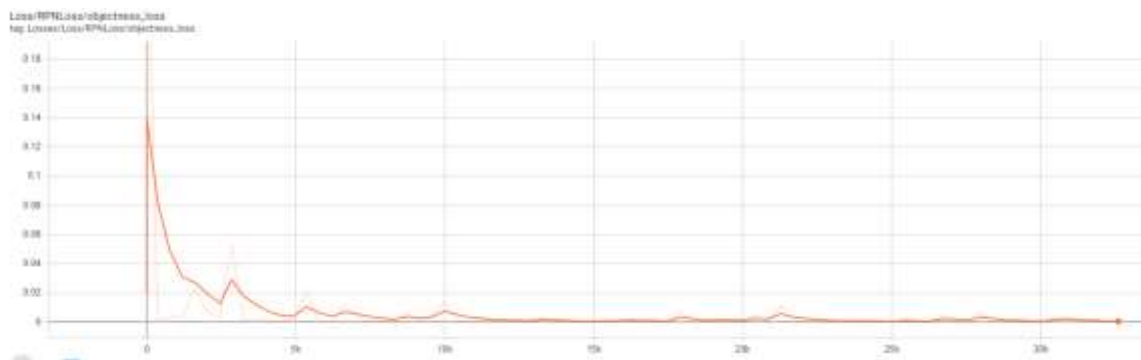
(Fig.25 – TensorBoard Loss Followers)

The total loss function is the overall loss of the model. It is also the loss that is displayed in the console. This is also the model's manual trigger to when the training process is satisfactory. As mentioned before, for the Oxford configured Faster R-CNN with Inception-V2 architecture, the recommended total loss value for a trained model should be around 0.05. From Figure 26 below, it can be seen that the total loss started high at a value greater than 2, and it kept getting lower until it consistently remained under 0.05, when the training was stopped after more than 30 thousand steps.



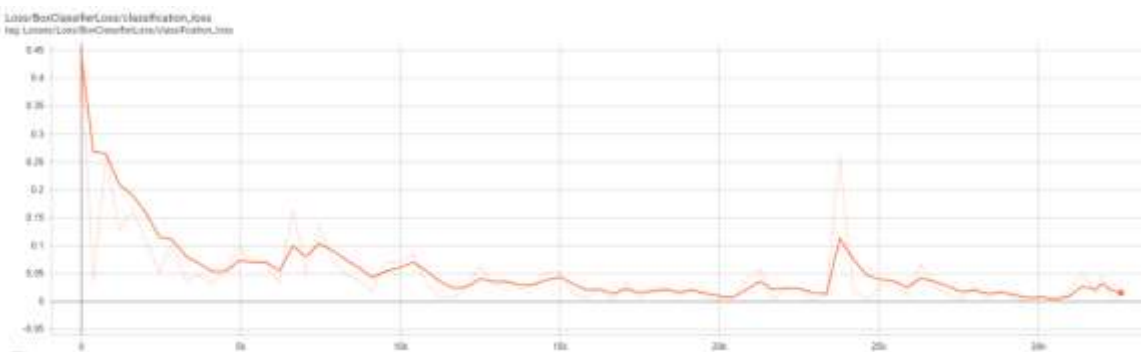
(Fig.26 – Total Loss Function plot)

The objective loss function plot shows the optimization done by the trainer. Basically, a probability for the change of getting to a defined objective function, where that objective function is the predefined correct answer is used for reference in the optimization process by the trainer. The objectiveness loss is also a type of RPN loss, which basically is the sum of the classification loss and the bounding box regression loss [12]. Figure 27 below shows exactly that, where the said offset can be seen starting from a value of 0.14 (Which in the first version of the model was much higher at around 0.6; more on that later)



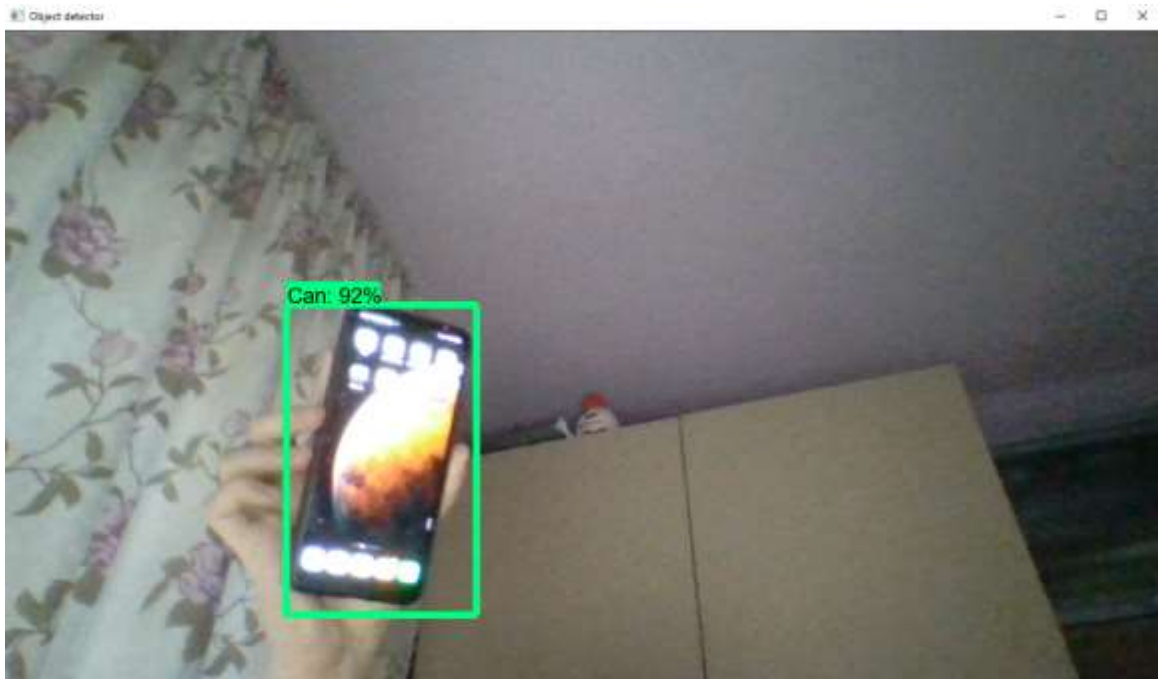
(Fig.27 – Objectiveness Loss Function plot)

And finally, the classifier loss. Basically, it uses cross-entropy loss to punish misclassified boxes, where cross-entropy loss (Log loss) measures the performance of the classifier with a value from 0 to 1 representing the predicated probability of divergence from the preferred result [12]. The final classifier loss plot can be seen in Figure 28 below.



(Fig.28 – Classifier Loss Function plot)

Currently the model is on its second version as in the first there were only around 300 images, which resulted in a partially bad model. It had very bad results for obturation, such as a can obturated by fingers or grass, it had varying success with detecting cans it never encountered and it gave many false positives on objects that it should not even consider as a possibility, such as humans. An example of such a false positive can be seen in Figure 29 below this paragraph. This was a direct cause of the training data. The first problem was the low variety of the test data which numbered at 62, which was not as well separated as in the current version. In the current version images are separated proportionally by situation, meaning that if there are 10 images of cans in grass, 2 go to the testing images while 8 go to the training images, holding the 20% rule discussed in Section 5.2. The second problem was the number of images and the range of new situations they contain. “A good object detector to improve when supplied with more training data ... [but it] saturated after only a few hundred to a thousand training examples” [13]. Based on this, in the first version, the saturation point was not reached and for this reason the team focused on optimizing the model by adding another 500 images which focused on all the problem areas. In the end, the model provided satisfactory results, which are covered in the next section.



(Fig.29 – Example of the faulty side of the first version of the model before optimization)

6. Results

In this section, the results of the whole project are taken and quantified in order to have a better understanding of the efficiency of the neural network, what are its strengths and what are its weaknesses.

6.1. General Results

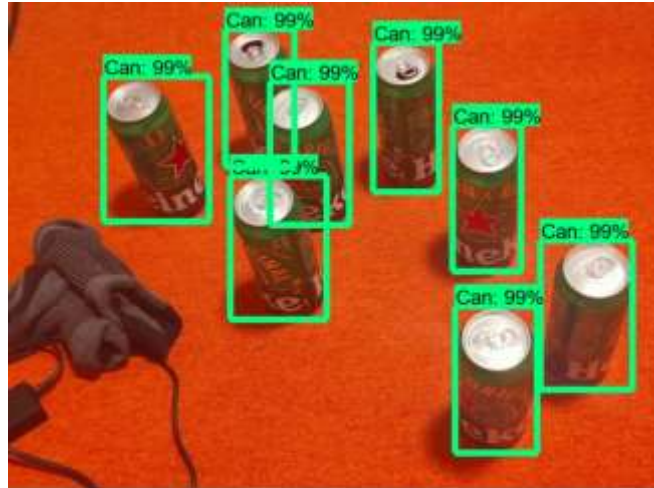
After the training and optimisation process, the neural network was in a good enough spot for the team to start focusing on recording the results. As a general set of results, the neural network had no problem detecting perfect cans and deformed cans, it had no problem with obscuration (such as a can held in hand or a can in grass), but it had some problems with throwing false positives on objects that resemble cans and it hasn't met in the training process. This could be solved in the future by adding a big database to the custom one the team created, which contains no cans, in order to teach the neural network what isn't a can rather than what is a can. Some snapshots of the neural network in action can be seen in Figures 30, 31, 32 and 33 below.



(Fig.30 – Snapshots of perfect and deformed cans)



(Fig.31 – Snapshot of a can obscured by grass)



(Fig.32 – Snapshot of the can detector testing cans in different states)



(Fig.33 – Snapshot of cans obscured by hands)

6.2. Quantifying the results

In order to get a more conclusive answer on the performance of the neural network, the team came up with a way to quantify the results. For this a few things had to be tested:

- NN performance on perfect cans
- NN performance on deformed cans
- NN performance with obscuration
- NN false positives

The first three bullet points were measured using 100 pictures that encapsulate them. It should be noted that the brands of the cans in the 100 pictures were different to the ones used in training. In order to get a constant output result, rather than using video feed to

check everything, the pictures were fed into the NN one at a time and the results were recorded in the Tables 1 and 2, which can be seen in the Appendix because of their size. Even so, the results are summarised in Table 3 below:

State	Score
Perfect cans detection	86%
..of which obturated	95.55%
Deformed cans detection	94.23%
..of which obturated	90%
Overall can detection	90.12%
Overall obturation detection	94.54%

(Table 3 – Condensed results)

As an extra to the plain quantification of the results, some notes were taken in scenarios that were out of expectations in order to better understand the NN’s “thinking” process. The most prominent result was the multiple bounding boxes containing each other. For these the score was not penalised as long as they were correct as this could be a simple fix for the next version of the neural network in which the display of a bounding box by a percentage of intersection will be conditioned (For example, if less than 60% of the bounding box is not intersecting another bounding box, display it, if not, do not). It was also seen that the neural network has problems with: cans that have an aluminium covering over the top side (Not enough training data on the situation), close-ups of cans that are covered with moisture (It “distorts” the shape of the can. More training data needed covering this case) and weird can designs (horizontal strips, caricatures of cans, images of cans on cans, etc.).

Not that the performance of the neural network was quantified, the false positives need to be recorded. Because the project is mostly focused on a waste sorter/detector focused on cans, images of recycled waste were used. In order to get consistent results, the images were made from a perpendicular angle on tarmac as it provides a high contrast background. Once again multiple images were passed through the neural network in order to form Table 4 below containing the most common false positives.

Most Common False Positives
Egg carton (6 pack)
Honey bottle
Vitamin bottle
Ice cream cup
Bailey’s bottle
Sauce bottle
Bottle cap
Beer bottle
Plastic tray
Small cream bottle
Plastic cup
Glass jar
Hot chocolate container
Battery
Plastic lid
Soup carton

(Table 4 – False Positives)

From the information collected in Table 4, a common theme is present. It has problems differentiating between cans, recipients and some cylindrical objects. This is a direct cause of the fact that there weren't enough "not-a-can" scenarios for the Neural Network to learn from. But just as mentioned in the previous sentence, this could be solved by improving the current dataset with many images that do not contain cans in order for it to have a broader understanding of new encounters and also add images of common containers/recipients and cylindrical object in order to emphasize them as not being cans.

6.3. Final Results

The can detector can be summarized as having a 90.12% overall accuracy in detecting cans and a 94.54% accuracy in detecting obturated cans. It also had good results for deformity, where it scored a 94.23% overall accuracy; all the values being taken from Table 3, section 6.2.

With an overall accuracy of over 90% and a very good understanding of the difficulties of the current model along with their fixes, the can detector can be called a success but there is room for improvement. These small mistakes can be easily fixed with time and small adjustments. Some steps that can be taken to improve the neural network in the future are listed below:

- Add to the image dataset pictures of cans covered in moisture
- Add to the image dataset a wider variety of cans
- Add to the image dataset many pictures that do not contain cans in order to act like a "not-a-can" scenario to reduce false positives.
- Create a conditioner for the bounding box to not cover the same objects.
- Add to the image dataset pictures of cylindrical objects that are not cans

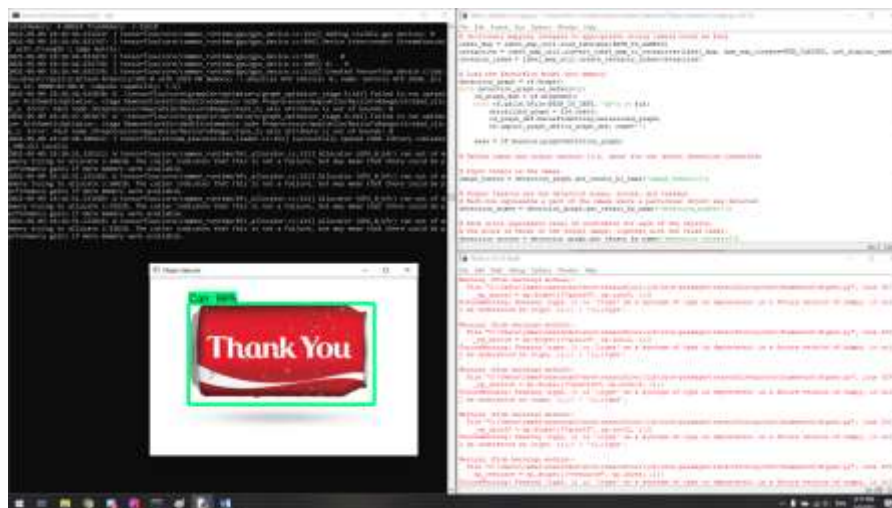
In the end, with around a 90% accuracy rate, the can detector can be called a general success as it detects cans with easy in all states of deformation and obturation. While it has some problems with false calls, those are well understood, and improvements can be easily done in the future to correct them.

7. Conclusion

The team started with barely any info on neural networks, let alone on how to create or use one. For the first few weeks the main task was reading on how neural networks work, how to create them, how to use them and what the options out there are. One of the first choices made was to use an existing architecture, as there was no reason for the team to attempt reinventing the wheel instead of investing its time in the better understanding of neural networks, optimization methods and dataset creation. For this reason, the TensorFlow API was chosen. Another choice that was made in order to save as much of the limited time provided, was the use of GPU training. For this CUDA and cuDNN were installed as well. After that the open source TensorFlow modeller was taken and adapted for the can detector implementation and with the 800 pictures that were taken, labelled and separated, the training began. After around 30000 steps and 3 hours of waiting, the neural network was trained, and it offered results over the team's expectation.

The can detector has an accuracy of over 90.12% including perfect cans, deformed cans and obturated cans. While there is an error margin of 9.88% based on the tests conducted, the causes are known and the respective fixes were thought of, which in the future can improve the can detector to an even better accuracy. Even so, the trained model can easily detect cans in all states of deformation and obturation, and as such it can be concluded that the project was a success.

Along with the can detector's success, the project also managed to provide the team members with good experience with neural network theory and the TensorFlow implementation of one, which will provide a strong foundation for any future projects. It also managed to introduce us to Python, which eased our introduction in the programming language providing the basics. Figure 34 below this paragraph provides the ending note of this project.



(Fig.34 – Thank you!)

8. References

- [1] A. Thomas, "An introduction to neural networks for beginners".
- [2] Y. Wu and J. Feng, "Development and Application of Artificial Neural Network," *Wireless Pers Commun*, vol. 102, p. 1645–1656, 2017.
- [3] P. Benardos and V. G.-C, "Engineering Applications of Artificial Intelligence," *Optimizing feedforward artificial neural network architecture*, vol. 20, no. 3, pp. 365-382, 2006.
- [4] J. Kolbusz, P. Rozycki and B. M. Wilamowski, "The Study of Architecture MLP with Linear Neurons in Order to Eliminate the “vanishing Gradient” Problem," *Artificial Intelligence and Soft Computing*, pp. 97-106, 2017.
- [5] Google Brain Team, "TensorFlow: A System for Large-Scale," *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pp. 271-272, 2016.
- [6] G. Van Houdt, C. Mosquera and G. Nápoles, "A review on the long short-term memory model," *Artificial Intelligence Review*, vol. 53, no. 8, pp. 5929-5955, 2020.
- [7] M. Sun, Z. Song, X. Jiang, J. Pan and Y. Pang, "Learning Pooling for Convolutional Neural Network," *Neurocomputing*, vol. 224, pp. 96-104, 2017.
- [8] T. Guo, J. Dong, H. Li and Y. Gao, "Simple convolutional neural network on image classification," *IEEE 2nd International Conference on Big Data Analysis (ICBDA)*, 2017.
- [9] M. A. Neilson, *Neural Networks and Deep Learning*, Determination Press, 2015.
- [10] B. Ripley, *Pattern Recognition and Neural Networks*, Cambridge University Press, 1996.
- [11] Y. B. a. A. C. Ian Goodfellow, *Deep Learning*, The MIT Press, 2016.
- [12] K. H. R. G. a. J. S. Shaoqing Ren, "Faster R-CNN: Towards Real-Time Object," 2016.
- [13] R. Girshick, "Fast R-CNN," *IEEE*, Santiago, Chile, 2015.

9. Appendix

In order for everything to be a little bit more organized, the appendix is separated into three parts that contain either the code or the large tables used for result determination.

9.1.Tables

Img.	Obtured	Perfect/Deformed	No. of cans	No. detected	Score	Notes
1	y	p	1	1	100	
2	y	p	2	2	200	
3	y	p	2	2	200	
4	y	p	1	2	100	box within box
5	y	p	2	2	200	
6	n	p	2	2	200	
7	y	p	1	1	100	
8	y	p	1	0	0	
9	n	p	1	1	100	
10	y	p	5	5	500	
11	y	p	2	2	200	
12	n	p	1	1	100	
13	y	p	4	3	350	two cans in one box
14	y	p	1	1	100	
15	y	p	1	1	100	
16	n	p	1	1	100	
17	y	p	3	3	300	
18	n	p	1	1	100	
19	n	p	1	1	100	
20	n	p	3	3	300	
21	n	p	1	1	100	
22	y	p	4	4	400	
23	y	p	2	2	200	
24	n	p	1	1	100	
25	y	p	8	5	500	
26	n	p	1	1	100	
27	n	p	1	1	100	
28	n	p	1	1	100	
29	n	p	1	1	100	
30	n	p	1	1	100	
31	n	p	1	1	100	

32	n	p	1	1	100	
33	n	p	3	2	200	
34	n	p	12	8	800	moisture/cap?
35	n	p	1	1	100	
36	n	p	4	4	400	
37	n	p	4	3	300	strips?
38	n	p	2	0	0	moisture breaks lines
39	n	p	2	1	100	doesn't look real
40	y	p	3	3	300	
41	n	p	1	1	100	
42	n	p	1	1	100	
43	n	p	1	1	100	
44	n	p	1	1	100	
45	n	p	1	1	100	
46	y	p	1	2	100	box within box
47	y	p	1	1	100	
48	y	p	1	1	100	
49	y	p	1	1	100	
50	y	p	1	1	100	
Number of Perfect Cans			100	Final Score	86	

(Table 1 – Results for detecting Perfect Cans)

Img.	Obtured	Perfect/Deformed	No. of cans	No. detected	Score	Notes
1	n	d	1	1	100	
2	n	d	1	2	100	box within box
3	n	d	1	1	100	
4	n	d	1	2	100	box within box
5	n	d	1	1	100	
6	n	d	1	2	100	box within box
7	n	d	1	1	100	
8	n	d	1	2	100	box within box
9	n	d	1	1	100	
10	n	d	1	1	100	
11	n	d	1	2	100	box within box
12	n	d	1	2	100	box within box
13	n	d	1	1	100	
14	n	d	2	1	100	
15	n	d	1	1	100	
16	n	d	1	1	100	
17	n	d	1	1	100	
18	n	d	1	1	100	
19	n	d	2	3	200	box within box

20	n	d	1	1	100	
21	n	d	1	1	100	
22	n	d	1	2	100	box within box
23	n	d	1	1	100	
24	n	d	1	1	100	
25	n	d	1	2	100	box within box
26	n	d	1	1	100	
27	n	d	0	0	0	
28	n	d	1	1	100	
29	n	d	1	1	100	
30	n	d	1	3	100	box within box
31	n	d	1	1	100	
32	n	d	1	2	100	
33	n	d	1	2	100	box within box
34	n	d	2	2	200	
35	n	d	1	1	100	
36	n	d	1	2	100	box within box
37	n	d	1	0	0	
38	n	d	1	1	100	
39	n	d	1	2	100	
40	n	d	1	1	100	
41	y	d	1	1	100	
42	y	d	1	0	0	
43	y	d	1	1	100	
44	y	d	1	1	100	
45	y	d	1	1	100	
46	y	d	1	2	100	box within box
47	y	d	1	2	100	
48	y	d	1	1	100	tab
49	y	d	1	1	100	
50	y	d	1	1	100	
Number of Deformed cans			52	Total score	94.23	

(Table 2 – Results for detecting Deformed Cans)

9.2. Python Script – Pictures

```
##### Image Object Detection Using Tensorflow-trained Classifier #####
#
# Description:
# This program uses a TensorFlow-trained neural network to perform object detection.
# It loads the classifier and uses it to perform object detection on an image.
# It draws boxes, scores, and labels around the objects of interest in the image.

## Some of the code is based on the open-source example provided by Google for the
## tensorflow image classifier
## and it also has influences from the EdgeElectronics application of a Neural Network
## Classifier. Both links
## can be seen below:
## https://github.com/datitran/object_detector_app/blob/master/object_detection_app.py
## https://github.com/EdgeElectronics/TensorFlow-Object-Detection-API-Tutorial-Train-
Multiple-Objects-Windows-10

# Import packages
import os
import cv2
import numpy as np
import tensorflow as tf
import sys

# This is needed since the notebook is stored in the object_detection folder.
sys.path.append("..")

# Import utilites
from utils import label_map_util
from utils import visualization_utils as vis_util

# Name of the directory containing the object detection module we're using
MODEL_NAME = 'inference_graph'
IMAGE_NAME = 'Testing the NN/Others/10.jpg'

# Grab path to current working directory
CWD_PATH = os.getcwd()

# Path to frozen detection graph .pb file, which contains the model that is used
# for object detection.
PATH_TO_CKPT = os.path.join(CWD_PATH, MODEL_NAME, 'frozen_inference_graph.pb')

# Path to label map file
PATH_TO_LABELS = os.path.join(CWD_PATH, 'training', 'labelmap.pbtxt')

# Path to image
PATH_TO_IMAGE = os.path.join(CWD_PATH, IMAGE_NAME)

# Number of classes the object detector can identify
NUM_CLASSES = 6

# Load the label map.
# Label maps map indices to category names, so that when our convolution
# network predicts `5`, we know that this corresponds to `king`.
# Here we use internal utility functions, but anything that returns a
# dictionary mapping integers to appropriate string labels would be fine
label_map = label_map_util.load_labelmap(PATH_TO_LABELS)
categories = label_map_util.convert_label_map_to_categories(label_map,
max_num_classes=NUM_CLASSES, use_display_name=True)
```

```

category_index = label_map_util.create_category_index(categories)

# Load the Tensorflow model into memory.
detection_graph = tf.Graph()
with detection_graph.as_default():
    od_graph_def = tf.GraphDef()
    with tf.gfile.GFile(PATH_TO_CKPT, 'rb') as fid:
        serialized_graph = fid.read()
        od_graph_def.ParseFromString(serialized_graph)
        tf.import_graph_def(od_graph_def, name='')

    sess = tf.Session(graph=detection_graph)

# Define input and output tensors (i.e. data) for the object detection classifier

# Input tensor is the image
image_tensor = detection_graph.get_tensor_by_name('image_tensor:0')

# Output tensors are the detection boxes, scores, and classes
# Each box represents a part of the image where a particular object was detected
detection_boxes = detection_graph.get_tensor_by_name('detection_boxes:0')

# Each score represents level of confidence for each of the objects.
# The score is shown on the result image, together with the class label.
detection_scores = detection_graph.get_tensor_by_name('detection_scores:0')
detection_classes = detection_graph.get_tensor_by_name('detection_classes:0')

# Number of objects detected
num_detections = detection_graph.get_tensor_by_name('num_detections:0')

# Load image using OpenCV and
# expand image dimensions to have shape: [1, None, None, 3]
# i.e. a single-column array, where each item in the column has the pixel RGB value
image = cv2.imread(PATH_TO_IMAGE)
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
image_expanded = np.expand_dims(image_rgb, axis=0)

# Perform the actual detection by running the model with the image as input
(boxes, scores, classes, num) = sess.run(
    [detection_boxes, detection_scores, detection_classes, num_detections],
    feed_dict={image_tensor: image_expanded})

# Draw the results of the detection (aka 'visulaize the results')

vis_util.visualize_boxes_and_labels_on_image_array(
    image,
    np.squeeze(boxes),
    np.squeeze(classes).astype(np.int32),
    np.squeeze(scores),
    category_index,
    use_normalized_coordinates=True,
    line_thickness=8,
    min_score_thresh=0.95)

# All the results have been drawn on image. Now display the image.
cv2.imshow('Object detector', image)

# Press any key to close the image
cv2.waitKey(0)

# Clean up
cv2.destroyAllWindows()

```

9.3. Python Script – Webcam

```
##### Webcam Object Detection Using Tensorflow-trained Classifier #####
#
# Description:
# This program uses a TensorFlow-trained classifier to perform object
# detection.
# It loads the classifier and uses it to perform object detection on a webcam # feed.
# It draws boxes, scores, and labels around the objects of interest in each
# frame from the webcam.

## Some of the code is based on the open-source example provided by Google
## for the tensorflow image classifier
## and it also has influences from the EdjeElectronics application of a
## Neural Network Classifier. Both links
## can be seen below:
## https://github.com/datitran/object_detector_app/blob/master/object_detection_app.py
## https://github.com/EdjeElectronics/TensorFlow-Object-Detection-API-Tutorial-Train-
Multiple-Objects-Windows-10

# Import packages
import os
import cv2
import numpy as np
import tensorflow as tf
import sys

# This is needed since the notebook is stored in the object_detection folder.
sys.path.append("..")

# Import utilites
from utils import label_map_util
from utils import visualization_utils as vis_util

# Name of the directory containing the object detection module we're using
MODEL_NAME = 'inference_graph'

# Grab path to current working directory
CWD_PATH = os.getcwd()

# Path to frozen detection graph .pb file, which contains the model that is used
# for object detection.
PATH_TO_CKPT = os.path.join(CWD_PATH, MODEL_NAME, 'frozen_inference_graph.pb')

# Path to label map file
PATH_TO_LABELS = os.path.join(CWD_PATH, 'training', 'labelmap.pbtxt')

# Number of classes the object detector can identify
NUM_CLASSES = 1

## Load the label map.
label_map = label_map_util.load_labelmap(PATH_TO_LABELS)
categories = label_map_util.convert_label_map_to_categories(label_map,
max_num_classes=NUM_CLASSES, use_display_name=True)
category_index = label_map_util.create_category_index(categories)

# Load the Tensorflow model into memory.
detection_graph = tf.Graph()
with detection_graph.as_default():
```

```

od_graph_def = tf.GraphDef()
with tf.gfile.GFile(PATH_TO_CKPT, 'rb') as fid:
    serialized_graph = fid.read()
    od_graph_def.ParseFromString(serialized_graph)
    tf.import_graph_def(od_graph_def, name='')

sess = tf.Session(graph=detection_graph)

# Define input and output tensors (i.e. data) for the object detection classifier

# Input tensor is the image
image_tensor = detection_graph.get_tensor_by_name('image_tensor:0')

# Output tensors are the detection boxes, scores, and classes
# Each box represents a part of the image where a particular object was detected
detection_boxes = detection_graph.get_tensor_by_name('detection_boxes:0')

# Each score represents level of confidence for each of the objects.
# The score is shown on the result image, together with the class label.
detection_scores = detection_graph.get_tensor_by_name('detection_scores:0')
detection_classes = detection_graph.get_tensor_by_name('detection_classes:0')

# Number of objects detected
num_detections = detection_graph.get_tensor_by_name('num_detections:0')

# Initialize webcam feed
video = cv2.VideoCapture(0)
ret = video.set(3,1280)
ret = video.set(4,720)

while(True):
    # Acquire frame and expand frame dimensions to have shape: [1, None, None, 3]
    # i.e. a single-column array, where each item in the column has the pixel RGB
    value
    ret, frame = video.read()
    frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    frame_expanded = np.expand_dims(frame_rgb, axis=0)

    # Perform the actual detection by running the model with the image as input
    (boxes, scores, classes, num) = sess.run(
        [detection_boxes, detection_scores, detection_classes, num_detections],
        feed_dict={image_tensor: frame_expanded})

    # Draw the results of the detection (aka 'visualize the results')
    vis_util.visualize_boxes_and_labels_on_image_array(
        frame,
        np.squeeze(boxes),
        np.squeeze(classes).astype(np.int32),
        np.squeeze(scores),
        category_index,
        use_normalized_coordinates=True,
        line_thickness=8,
        min_score_thresh=0.75)

    # All the results have been drawn on the frame, so it's time to display it.
    cv2.imshow('Object detector', frame)

    # Press 'q' to quit
    if cv2.waitKey(1) == ord('q'):
        break

```

```
# Clean up
video.release()
cv2.destroyAllWindows()
```